Sri Indu
College of Engineering & Technology
UGC Autonomous Institution
Recognized under 2(f) & 12(B) of UGC Act 1956,
NAAC, Approved by AICTE &
Permanently Affiliated to JNTUH

Estd.2001

NAAC
NATIONAL ASSESSMENT AND ACCREDITATION COUNCIL

OS LAB MANUAL

II Year-I Semester

DEPARTMENT OF ARTIFICIAL INTELLIGENCE & DATA SCIENCE

ACADEMIC YEAR -2025-2026

SRI INDU COLLEGE OF ENGINEERING & TECHNOLOGY
(An Autonomous Institution under UGC, New Delhi)
Recognized under 2(f) and 12(B) of UGC Act 1956
NBA Accredited, Approved by AICTE and Permanently affiliated to JNTUH
Sheriguda (V), Ibrahimpatnam, R.R.Dist, Hyderabad - 501 510

Department of Artificial Intelligence & Data Science

LAB MANUAL

Branch: AI&DS **Class: B.Tech- II Year-I sem**

Subject: OPERATING SYSTEMS LAB **Code: R22CSE2226**

Academic Year: 2025-26 **Regulation: R22**

Prepared By

Name: Mrs. R.Madhava Reddy(Asst. Prof.)

Verified By

Head of the Department

SRI INDU COLLEGE OF ENGINEERING & TECHNOLOGY

(An Autonomous Institution under UGC, New Delhi)

B.Tech. - II Year – I Semester

L	T	P	C
0	0	2	1

(R22CSE2226) OPERATING SYSTEMS LAB

Course Objectives:

- To provide an understanding of the design aspects of operating system concepts through simulation
- Introduce basic Unix commands, system call interface for process management, interprocess communication and I/O in Unix

Course Outcomes:

● Simulate and implement operating system concepts such as scheduling, deadlockmanagement, file management and memory management.
● Able to implement C programs using Unix system calls

List of Experiments:
1. Write C programs to simulate the following CPU Scheduling algorithms a) FCFS b) SJF c) Round Robin d) priority
2. Write programs using the I/O system calls of UNIX/LINUX operating system (open, read, write, close, fcntl, seek, stat, opendir, readdir)
3. Write a C program to simulate Bankers Algorithm for Deadlock Avoidance and Prevention.
4. Write a C program to implement the Producer – Consumer problem using semaphores using UNIX/LINUX system calls.
5. Write C programs to illustrate the following IPC mechanisms a) Pipes b) FIFOs c) Message Queues d) Shared Memory
6. Write C programs to simulate the following memory management techniques a) Paging b) Segmentation
7. Write C programs to simulate Page replacement policies a) FCFS b) LRU c) Optimal

TEXT BOOKS:

1. Operating System Principles- Abraham Silberchatz, Peter B. Galvin, Greg Gagne 7 th Edition, John Wiley

2. Advanced programming in the Unix environment, W.R.Stevens, Pearson education.

REFERENCE BOOKS:

1. Operating Systems – Internals and Design Principles, William Stallings, Fifth Edition–2005, Pearson Education/PHI

2. Operating System - A Design Approach-Crowley, TMH.

3. Modern Operating Systems, Andrew S Tanenbaum, 2nd edition, Pearson/PHI

4. UNIX Programming Environment, Kernighan and Pike, PHI/Pearson Education

5. UNIX Internals: The New Frontiers, U. Vahalia, Pearson Education

EXP1:

1. Write C programs to simulate the following CPU Scheduling algorithms

a) FCFS

1. Aim

To implement the **First Come First Serve (FCFS)** CPU scheduling algorithm in C, calculate **Waiting Time** (WT), **Turnaround Time** (TAT), and display results along with averages.

2. Theory

Definition:

FCFS is the simplest, non-preemptive scheduling algorithm where the process that arrives first is executed first.

- **Process executes in the order of arrival.**
- Once a process starts, it runs till completion.
- **Advantage:** Easy to implement
- **Disadvantage:** May cause **Convoy Effect** when long processes delay others.

Terminology

- **Burst Time (BT):** CPU time required by a process.
- **Arrival Time (AT):** Time when the process enters the ready queue.
- **Completion Time (CT):** Time when the process finishes execution.
- **Turnaround Time (TAT):**

$$TAT = CT - AT$$

- **Waiting Time (WT):**

$$WT = TAT - BT$$

3. Mathematical Model

For n processes:

1. **Waiting Time:**
   - $WT_1 = 0$
   - $WT[i] = WT[i-1] + BT[i-1]$ for $i \geq 2$
2. **Turnaround Time:**
   - $TAT[i] = WT[i] + BT[i]$
3. **Averages:**

$$AvgWT = \frac{\sum WT}{n}$$

$$AvgTAT = \frac{\sum TAT}{n}$$

4. Example Calculation

Given:

Process	BT	AT	WT	TAT
P1	5	0	0	5
P2	3	0	5	8
P3	8	0	8	16

- Avg WT = (0 + 5 + 8) / 3 = **4.33**
- Avg TAT = (5 + 8 + 16) / 3 = **9.67**

5. Algorithm

1. Input the number of processes and their **burst times**.
2. Set WT for first process = 0.
3. Calculate WT for remaining processes using cumulative BT.
4. Calculate TAT = WT + BT for each process.
5. Compute average WT and average TAT.
6. Display results.

6.C Program:

```
#include <stdio.h>
int main() { int
    n, i;
    int burst_time[20], waiting_time[20], turnaround_time[20]; int total_wt
    = 0, total_tat = 0;
    printf("Enter number of processes: "); scanf("%d",
    &n);
    printf("Enter burst time for each process:\n"); for(i =
    0; i < n; i++) {
        printf("P[%d]: ", i + 1);
        scanf("%d", &burst_time[i]);
    }
    waiting_time[0] = 0; // First process waiting time = 0

    for(i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i - 1] + burst_time[i - 1];
    }
    for(i = 0; i < n; i++) {
        turnaround_time[i] = waiting_time[i] + burst_time[i];
```

```
    total_wt += waiting_time[i];          total_tat
     += turnaround_time[i];
      }

      printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n"); for(i = 0; i < n;
      i++) {
          printf("P[%d]\t%d\t\t%d\t\t%d\n", i + 1, burst_time[i], waiting_time[i],
turnaround_time[i]);
      }

      printf("\nAverage Waiting Time = %.2f\n", (float)total_wt / n); printf("Average
      Turnaround Time = %.2f\n", (float)total_tat / n);
      return 0;
}
```

7.Output:

```
Enter number of processes: 3 Enter
burst time for each process: P[1]: 5
P[2]: 3
P[3]: 8
```

Process	Burst Time	Waiting Time	Turnaround Time
P[1]	5	0	5
P[2]	3	5	8
P[3]	8	8	16

Average Waiting Time = 4.33 Average

Turnaround Time = 9.67

8.Execution Flow:

| P1 | -----5-----| P2 | -----3-----| P3 | ----------8----------|

0 58 16

9.Conclusion:

- FCFS executes processes in arrival order.
- It is simple but can cause delays if a large process arrives first (Convoy Effect).
- Best suited for batch processing systems.

1.Write C programs to simulate the following CPU Scheduling algorithms

B) SJF

C Program

```
#include <stdio.h>


int main() {
    int n, i, j, pos, temp;
    int burst_time[20], process[20], waiting_time[20], turnaround_time[20]; int total_wt =
    0, total_tat = 0;


    printf("Enter number of processes: "); scanf("%d",
    &n);


    printf("Enter burst time for each process:\n"); for(i =
    0; i < n; i++) {
        printf("P[%d]: ", i + 1);
        scanf("%d", &burst_time[i]);
        process[i] = i + 1; // Store process numbers
    }
```

```
// Sort burst times with process number using selection sort
 for(i = 0; i < n; i++) {
    pos = i;
    for(j = i+1; j < n; j++) { if(burst_time[j] <
        burst_time[pos])
            pos = j;
    }
    temp = burst_time[i]; burst_time[i] =
    burst_time[pos]; burst_time[pos] =
    temp;

    temp = process[i]; process[i]
    = process[pos]; process[pos]
    = temp;
}

// First process has 0 waiting time waiting_time[0] =
0;

// Calculate waiting time for(i =
1; i < n; i++) {
    waiting_time[i] = 0; for(j
    = 0; j < i; j++)
        waiting_time[i] += burst_time[j]; total_wt +=
    waiting_time[i];
}
```

```
    // Calculate turnaround time for(i =
    0; i < n; i++) {
        turnaround_time[i] = burst_time[i] + waiting_time[i]; total_tat +=
        turnaround_time[i];
    }

    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n"); for(i = 0; i < n;
    i++)
        printf("P[%d]\t%d\t\t%d\t\t%d\n", process[i], burst_time[i],
waiting_time[i], turnaround_time[i]);

    printf("\nAverage Waiting Time = %.2f", (float)total_wt / n); printf("\nAverage
    Turnaround Time = %.2f\n", (float)total_tat / n);

    return 0;
}
```

7. Sample Output

```
text
Enter number of processes: 4
Enter burst time for each process:
P[2]: 6
P[3]: 8
P[4]: 7
P[5]: 3

Process Burst Time Waiting Time Turnaround Time
P[5]    3          0            3
P[2]    6          3            9
P[4]    7          9            16
P[3]    8          16           24

Average Waiting Time = 7.00
Average Turnaround Time = 13.00
```

1.Write C programs to simulate the following CPU Scheduling algorithms

C)Round Robin

```
#include <stdio.h>
int main() { int
    n;
    printf("Enter Total Number of Processes: ");
    scanf("%d", &n);
    int wait_time = 0, ta_time = 0;
    int arr_time[n], burst_time[n], temp_burst_time[n]; int x = n;
    // number of remaining processes
    for (int i = 0; i < n; i++) {
        printf("Enter Details of Process %d\n", i + 1); printf("Arrival Time:
        ");
        scanf("%d", &arr_time[i]);
        printf("Burst Time: ");
        scanf("%d", &burst_time[i]);
        temp_burst_time[i] = burst_time[i];
    }
    int time_slot;
    printf("Enter Time Slot: ");
    scanf("%d", &time_slot);
```

```
    int total = 0, counter = 0, i = 0;
    printf("\nProcess ID  Burst Time  Turnaround Time  Waiting Time\n");

    while (x != 0) {
        if (temp_burst_time[i] <= time_slot && temp_burst_time[i] > 0) { total +=
            temp_burst_time[i];
            temp_burst_time[i] = 0;
            counter = 1;
        } else if (temp_burst_time[i] > 0) {
            temp_burst_time[i] -= time_slot; total
            += time_slot;
        }

        if (temp_burst_time[i] == 0 && counter == 1) { x--;
            printf("P%-9d %-11d %-15d %-10d\n",
                    i + 1, burst_time[i], total - arr_time[i], total - arr_time[i] -
burst_time[i]);
            wait_time += total - arr_time[i] - burst_time[i]; ta_time
            += total - arr_time[i];
            counter = 0;
        }
        if (i == n - 1) { i
            = 0;
        } else if (arr_time[i + 1] <= total) {
```

```
            i++;
        } else {
            i = 0;
        }
    }

    float average_wait_time = (float)wait_time / n; float
    average_turnaround_time = (float)ta_time / n;

    printf("\nAverage Waiting Time: %.2f", average_wait_time); printf("\nAverage
    Turnaround Time: %.2f\n", average_turnaround_time);

    return 0;
}
```

Output:

```
Enter Total Number of Processes: 3
Enter Details of Process 1
Arrival Time: 0
Burst Time: 5
Enter Details of Process 2
Arrival Time: 1
Burst Time: 4
Enter Details of Process 3
Arrival Time: 2
Burst Time: 2
Enter Time Slot: 2

Process ID  Burst Time  Turnaround Time  Waiting Time
P3          2           4                2
P2          4           9                5
P1          5           11               6

Average Waiting Time: 4.33
Average Turnaround Time: 8.00

--------------------------------
Process exited after 92.2 seconds with return value 0
Press any key to continue . . .
```

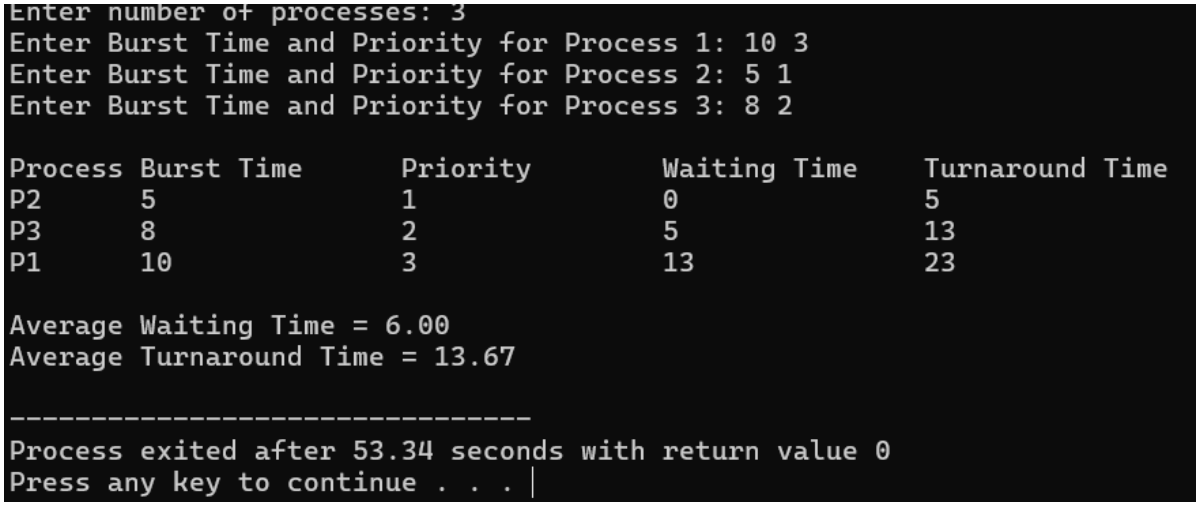
1.Write C programs to simulate the following CPU Scheduling algorithms
D)Priority

```
#include <stdio.h>

  int main() {

      int n, i, j, pos, temp;

      int burst_time[20], p[20], wait_time[20], tat[20], priority[20]; float

      total_wait = 0, total_tat = 0;


printf("Enter number of processes: "); scanf("%d", &n);

      for(i = 0; i < n; i++) {

          printf("Enter Burst Time and Priority for Process %d: ", i + 1); scanf("%d %d",

          &burst_time[i], &priority[i]);

          p[i] = i + 1;  // process ID

      }
      // Sort processes by priority (lower number = higher priority) for(i = 0; i

      < n; i++) {

          pos = i;

          for(j = i + 1; j < n; j++) { if(priority[j] <

              priority[pos])

                  pos = j;

          }

          // Swap priority
```

```
    temp = priority[i]; priority[i]
    = priority[pos]; priority[pos]
    = temp;

    // Swap burst time temp
    = burst_time[i];
    burst_time[i] = burst_time[pos]; burst_time[pos] =
    temp;
    // Swap process ID temp
    = p[i];
    p[i] = p[pos];
    p[pos] = temp;
}

// Calculate waiting times
wait_time[0] = 0;
for(i = 1; i < n; i++) {
    wait_time[i] = 0; for(j =
    0; j < i; j++)
        wait_time[i] += burst_time[j];
    total_wait += wait_time[i];
}

// Calculate turnaround time and print table
printf("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
```

```
    for(i = 0; i < n; i++) {
        tat[i] = burst_time[i] + wait_time[i]; total_tat +=
        tat[i];
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", p[i], burst_time[i], priority[i], wait_time[i],
tat[i]);
    }

    printf("\nAverage Waiting Time = %.2f", total_wait / n);
    printf("\nAverage Turnaround Time = %.2f\n", total_tat / n);

    return 0;
}
```

Output:



2.Write programs using the I/O system calls of UNIX/LINUX operating system (open, read, write, close, fcntl, seek, stat, opendir, readdir)

Aim: To perform I/O system calls of UNIX/LINUX operating system (open, read, write, close, fcntl, seek, stat, opendir, readdir)

/* Open GDB Online Debugger to Perform below Linux calls.--> Login First if required then perform all actions*/

/* 1.create a sample text file on pc desktop with name test.txt—usually notepad file and write some content called

Hello GDB Online Debugger.How are you.I am your friend */

/* Upload that file into GDB Online Debugger and execute the below program*/

/* On left side pane□Click □Create New Project□Write the prog□Execute*/

Program:

1. open(), read(), write(), close()

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
 int fd;
    char buffer[100];

    // Open file in read-only mode
    fd = open("test.txt", O_RDONLY);

    if (fd < 0) {
```

```
        perror("open"); return
        1;
    }

    // Read contents
    int n = read(fd, buffer, sizeof(buffer));

    if (n < 0) {
        perror("read");
        return 1;
    }
    buffer[n] = '\0';

    // Write contents to STDOUT write(1,
    buffer, n);

    // Close file
    close(fd);
     return 0;
}
```

Output:

2. fcntl() – File control

Program:

```
#include <stdio.h>

#include <fcntl.h> #

include <unistd.h>


int main() {
    int fd = open("test.txt", O_RDONLY);

    if (fd < 0) {
        perror("open");
        return 1;
    }


    // Get file descriptor flags
    int flags = fcntl(fd, F_GETFL);
    printf("File access mode: %d\n", flags & O_ACCMODE);


    close(fd);
     return 0;
}
```

Output:

3. lseek() – File offset positioning

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd = open("test.txt", O_RDONLY);
    char buffer[20];

    // Move file pointer 5 bytes ahead
    lseek(fd, 5, SEEK_SET);

    // Read after seeking
    int n = read(fd, buffer, 15);
    buffer[n] = '\0';
    printf("Data after seek: %s\n", buffer);

    close(fd);
    return 0;
    }
```

O/P:

4. stat() – File information

```
#include <stdio.h>
#include <sys/stat.h>

int main() {
    struct stat fileStat;

    if (stat("test.txt", &fileStat) < 0) {
        perror("stat");
        return 1;
    }

    printf("File Size: %ld bytes\n", fileStat.st_size);
    printf("Number of Links: %ld\n", fileStat.st_nlink);
    printf("File Permissions: %o\n", fileStat.st_mode & 0777);
    return 0;
}
```

O/P:

5.opendir(), readdir() – Directory operations

```
#include <stdio.h>
#include <dirent.h>

int main() {
    DIR *dir;
    struct dirent *entry;

    dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        return 1;
    }

    printf("Files in current directory:\n");
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }

    closedir(dir);
    return 0;
}
```

O/P:

3. Write a C program to simulate Bankers Algorithm for Deadlock Avoidance and Prevention.

This program checks whether the system is in a **safe state** using **Banker's Algorithm**.

C Program

```
#include <stdio.h>

int main() {
    int n, m, i, j, k;

    printf("Enter number of processes: ");
    scanf("%d", &n);

    printf("Enter number of resources: ");
    scanf("%d", &m);

    int alloc[n][m], max[n][m], need[n][m];
    int avail[m];

    printf("\nEnter Allocation Matrix:\n");
    for(i = 0; i < n; i++) {
        for(j = 0; j < m; j++) {
            scanf("%d", &alloc[i][j]);
        }
    }
    printf("\nEnter Maximum Matrix:\n");
    for(i = 0; i < n; i++) {
        for(j = 0; j < m; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    printf("\nEnter Available Resources:\n");
    for(i = 0; i < m; i++) {
        scanf("%d", &avail[i]);
    }
```

```
// Calculate Need Matrix
  for(i = 0; i < n; i++) {
    for(j = 0; j < m; j++) {
      need[i][j] = max[i][j] - alloc[i][j];
    }
  }

  int finish[n], safeSeq[n];
  for(i = 0; i < n; i++)
    finish[i] = 0;

  int count = 0;

  while(count < n) {
    int found = 0;

    for(i = 0; i < n; i++) {
      if(finish[i] == 0) {

        // Check if all resources can be allocated
        for(j = 0; j < m; j++) {
          if(need[i][j] > avail[j])
            break;
        }

        if(j == m) {
          // Process can execute
          for(k = 0; k < m; k++) {
            avail[k] += alloc[i][k];
          }
          safeSeq[count++] = i;
          finish[i] = 1;
          found = 1;
        }
      }
    }
    // No process could be executed
    if(found == 0) {
      printf("\nSystem is NOT in safe state (Deadlock may occur)\n");
      return 0;
    }
  }
```

```
 // Safe sequence
    printf("\nSystem is in SAFE state.\nSafe Sequence:\n");

    for(i = 0; i < n; i++) {
        printf("P%d", safeSeq[i]);

        if(i != n - 1)
            printf(" -> ");
    }

    printf("\n");

    return 0;
}
```

Input:
Enter number of processes: 5
Enter number of resources: 3

Enter Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2

Enter Maximum Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

Enter Available Resources:
3 3 2

Output:

```
System is in SAFE state.

Safe Sequence:
P1 -> P3 -> P4 -> P0 -> P2
```

4. Write a C program to implement the Producer – Consumer problem using semaphores using UNIX/LINUX system calls.

This program uses:

- **POSIX Threads (`pthread`)**
- **Semaphores (`sem_t`)**
- UNIX/Linux synchronization system calls.

C Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

// Semaphores
sem_t empty;
sem_t full;
pthread_mutex_t mutex;

// Producer Function
void *producer(void *arg) {
   int item;

   for(int i = 1; i <= 10; i++) {

      item = rand() % 100;

      sem_wait(&empty);              // wait if buffer full
      pthread_mutex_lock(&mutex);    // enter critical section

      buffer[in] = item;
      printf("Producer produced: %d\n", item);

      in = (in + 1) % BUFFER_SIZE;
```

```
  pthread_mutex_unlock(&mutex);  // leave critical section
        sem_post(&full);              // increase full count

        sleep(1);
    }

    pthread_exit(NULL);
}

// Consumer Function
void *consumer(void *arg) {
    int item;

    for(int i = 1; i <= 10; i++) {

        sem_wait(&full);              // wait if buffer empty
        pthread_mutex_lock(&mutex);    // enter critical section

        item = buffer[out];
        printf("Consumer consumed: %d\n", item);

        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);  // leave critical section
        sem_post(&empty);             // increase empty count

        sleep(2);
    }

    pthread_exit(NULL);
}

int main() {

    pthread_t p_thread, c_thread;

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
```

```
 // Initialize mutex
    pthread_mutex_init(&mutex, NULL);

    // Create producer and consumer threads
    pthread_create(&p_thread, NULL, producer, NULL);
    pthread_create(&c_thread, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(p_thread, NULL);
    pthread_join(c_thread, NULL);

    // Destroy semaphores and mutex
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

O/P:

```
Producer produced: 83
Consumer consumed: 83

Producer produced: 25
Producer produced: 67
Consumer consumed: 25

Producer produced: 91
Consumer consumed: 67

Producer produced: 12
Consumer consumed: 91

Producer produced: 48
Consumer consumed: 12
...
```

5.Write C programs to illustrate the following IPC mechanisms a) Pipes b) FIFOs c) Message Queues d) Shared Memory

(a) PIPE Program

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {

    int fd[2];
    char write_msg[] = "Hello through Pipe";
    char read_msg[50];

    pipe(fd);

    if(fork() == 0) {
        // Child Process
        close(fd[1]); // close write end
        read(fd[0], read_msg, sizeof(read_msg));
        printf("Child received: %s\n", read_msg);
    }
    else {
        // Parent Process
        close(fd[0]); // close read end
        write(fd[1], write_msg, strlen(write_msg)+1);
    }

    return 0;
}
```

O/P:

Child received: Hello through Pipe

(b) FIFO (Named Pipe)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main() {

    char *fifo = "myfifo";
    mkfifo(fifo, 0666);

    int fd;
    char arr1[] = "Hello FIFO";
    char arr2[100];

    if(fork() == 0) {
        // Child reads
        fd = open(fifo, O_RDONLY);
        read(fd, arr2, sizeof(arr2));
        printf("Received: %s\n", arr2);
        close(fd);
    }
    else {
        // Parent writes
        fd = open(fifo, O_WRONLY);
        write(fd, arr1, strlen(arr1)+1);
        close(fd);
    }

    return 0;
}
```

O/P:

Received: Hello FIFO

(c) Message Queue

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

struct msg_buffer {
    long msg_type;
    char msg_text[100];
} message;

int main() {

    key_t key;
    int msgid;

    key = ftok("progfile", 65);

    msgid = msgget(key, 0666 | IPC_CREAT);

    message.msg_type = 1;

    printf("Enter message: ");
    fgets(message.msg_text, sizeof(message.msg_text), stdin);

    msgsnd(msgid, &message, sizeof(message), 0);

    printf("Message sent.\n");

    msgrcv(msgid, &message, sizeof(message), 1, 0);

    printf("Message received: %s\n", message.msg_text);

    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

O/P:
Enter message: Hello IPC
Message sent.
Message received: Hello IPC

(d) Shared Memory

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {

    key_t key = 1234;

    int shmid = shmget(key, 1024, 0666 | IPC_CREAT);

    char *str = (char*) shmat(shmid, NULL, 0);

    printf("Write Data: ");
    gets(str);

    printf("Data written in memory: %s\n", str);

    shmdt(str);

    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

O/P:

Write Data: Operating System
Data written in memory: Operating System

Compilation Commands (Linux)

Pipe / FIFO

```
gcc pipe.c -o pipe
./pipe
```

Message Queue

```
gcc msgqueue.c -o msg
./msg
```

Shared Memory

```
gcc shm.c -o shm
./shm
```

6. Write C programs to simulate the following memory management techniques a) Paging b) Segmentation

a) C Program for Paging Technique

```
#include <stdio.h>

int main() {
    int pages, frames, page_size;
    int logical_address;
    int page_number, offset;

    printf("Enter page size: ");
    scanf("%d", &page_size);

    printf("Enter logical address: ");
    scanf("%d", &logical_address);

    // Calculate page number and offset
    page_number = logical_address / page_size;
    offset = logical_address % page_size;

    printf("\nPage Number = %d\n", page_number);
    printf("Offset = %d\n", offset);

    return 0;
}
```

O/P:

```
Enter page size: 100
Enter logical address: 245

Page Number = 2
Offset = 45
```

b) C Program for Segmentation Technique

```
#include <stdio.h>

int main() {
    int segment, offset;
    int base[5], limit[5];
    int physical_address;

    printf("Enter base and limit for 5 segments:\n");

    for(int i = 0; i < 5; i++) {
        printf("Segment %d Base: ", i);
        scanf("%d", &base[i]);

        printf("Segment %d Limit: ", i);
        scanf("%d", &limit[i]);
    }

    printf("\nEnter segment number: ");
    scanf("%d", &segment);

    printf("Enter offset: ");
    scanf("%d", &offset);

    // Check offset validity
    if(offset < limit[segment]) {
        physical_address = base[segment] + offset;

        printf("\nPhysical Address = %d\n", physical_address);
    }
    else {
        printf("\nSegmentation Fault! Offset exceeds limit.\n");
    }

    return 0;
}
```

O/P:

```
Enter base and limit for 5 segments:

Segment 0 Base: 1000
Segment 0 Limit: 500

Segment 1 Base: 2000
Segment 1 Limit: 300

Segment 2 Base: 3000
Segment 2 Limit: 400

Segment 3 Base: 4000
Segment 3 Limit: 600
```

Segment 4 Base: 5000
Segment 4 Limit: 700

Enter segment number: 2
Enter offset: 120

Physical Address = 3120

7.Write C programs to simulate Page replacement policies a) FCFS b) LRU c) Optimal

a) FCFS / FIFO Page Replacement Program

```
#include <stdio.h>

int main() {
   int pages[50], frames[10];
   int n, f, i, j, fault = 0, index = 0, found;

   printf("Enter number of pages: ");
   scanf("%d", &n);

   printf("Enter page reference string:\n");
   for(i = 0; i < n; i++)
      scanf("%d", &pages[i]);

   printf("Enter number of frames: ");
   scanf("%d", &f);

   for(i = 0; i < f; i++)
      frames[i] = -1;

   for(i = 0; i < n; i++) {

      found = 0;

      for(j = 0; j < f; j++) {
         if(frames[j] == pages[i]) {
            found = 1;
            break;
         }
      }

      if(found == 0) {
         frames[index] = pages[i];
         index = (index + 1) % f;
         fault++;
      }

      printf("\nFrames: ");
      for(j = 0; j < f; j++)
         printf("%d ", frames[j]);
   }

   printf("\n\nTotal Page Faults = %d\n", fault);

   return 0;
}
```

O/P:
Enter number of pages: 8
Enter page reference string:
1 2 3 4 1 2 5 1
Enter number of frames: 3

Frames: 1 -1 -1
Frames: 1 2 -1
Frames: 1 2 3
Frames: 4 2 3
Frames: 4 1 3
Frames: 4 1 2
Frames: 5 1 2
Frames: 5 1 2

Total Page Faults = 7

b) LRU Page Replacement Program

```
#include <stdio.h>

int main() {

   int pages[50], frames[10], time[10];
   int n, f, i, j, pos, fault = 0, count = 0;
   int found;

   printf("Enter number of pages: ");
   scanf("%d", &n);

   printf("Enter page reference string:\n");
   for(i = 0; i < n; i++)
      scanf("%d", &pages[i]);

   printf("Enter number of frames: ");
   scanf("%d", &f);

   for(i = 0; i < f; i++)
      frames[i] = -1;

   for(i = 0; i < n; i++) {

      found = 0;

      for(j = 0; j < f; j++) {
         if(frames[j] == pages[i]) {
            count++;
            time[j] = count;
            found = 1;
            break;
         }
      }

      if(found == 0) {

         pos = 0;

         for(j = 1; j < f; j++) {
            if(time[j] < time[pos])
               pos = j;
         }

         frames[pos] = pages[i];
         count++;
         time[pos] = count;
         fault++;
      }
```

```
printf("\nFrames: ");

for(j = 0; j < f; j++)
        printf("%d ", frames[j]);
    }

    printf("\n\nTotal Page Faults = %d\n", fault);

    return 0;
}
```

O/P:

```
Enter number of pages: 8
Enter page reference string:
1 2 3 4 1 2 5 1
Enter number of frames: 3

Frames: 1 -1 -1
Frames: 1 2 -1
Frames: 1 2 3
Frames: 4 2 3
Frames: 4 1 3
Frames: 4 1 2
Frames: 5 1 2
Frames: 5 1 2

Total Page Faults = 7
```

c) Optimal Page Replacement Program

```
#include <stdio.h>

int main() {

    int pages[50], frames[10];
    int n, f, i, j, k, fault = 0, found;

    printf("Enter number of pages: ");
    scanf("%d", &n);

    printf("Enter page reference string:\n");
    for(i = 0; i < n; i++)
        scanf("%d", &pages[i]);

    printf("Enter number of frames: ");
    scanf("%d", &f);

    for(i = 0; i < f; i++)
        frames[i] = -1;

    for(i = 0; i < n; i++) {

        found = 0;

        for(j = 0; j < f; j++) {
            if(frames[j] == pages[i]) {
                found = 1;
                break;
            }
        }

        if(found == 0) {

            int pos = -1, farthest = i + 1;

            for(j = 0; j < f; j++) {

                int found_future = 0;

                for(k = i + 1; k < n; k++) {
                    if(frames[j] == pages[k]) {
                        if(k > farthest) {
                            farthest = k;
                            pos = j;
                        }
                        found_future = 1;
                        break;
                    }
                }
```

```
    if(!found_future) {
                pos = j;
                break;
            }
        }

        if(pos == -1)
            pos = 0;

        frames[pos] = pages[i];
        fault++;
    }

    printf("\nFrames: ");
    for(j = 0; j < f; j++)
        printf("%d ", frames[j]);
    }

    printf("\n\nTotal Page Faults = %d\n", fault);

    return 0;
}
```

O/P:

```
Enter number of pages: 8
Enter page reference string:
1 2 3 4 1 2 5 1
Enter number of frames: 3

Frames: 1 -1 -1
Frames: 1 2 -1
Frames: 1 2 3
Frames: 1 2 4
Frames: 1 2 4
Frames: 1 2 4
Frames: 1 2 5
Frames: 1 2 5

Total Page Faults=5
```